A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX 4.2BSD

Luis Felipe Cabrera
Edward Hunter
Mike Karels
David Mosher

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA

INDEX

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
DEC 1984		**00-00-1984 to 00-00-1984**

4. TITLE AND SUBTITLE
A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX 4.2BSD

5a. CONTRACT NUMBER
5b. GRANT NUMBER
5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER
5e. TASK NUMBER
5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)
11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Berkeley UNIX 4.2BSD is an operating system which provides alternative ways for processes to communicate with each other. User processes may choose, for example, intermachine communications media, protocols, addressing families, and styles of communication. In particular, user processes may use datagram or stream communication. In this paper we present a study of the impact that IPC mechanisms, as currently implemented in Berkeley UNIX 4.2BSD in Ethernet based environments, have, from the user process viewpoint, on the performance of distributed applications. This study not only assesses the impact that different processors, network hardware interfaces, and Ethernets have on the communication across machines between user processes, but also the effect of the loading of the various components which participate in the interprocess communication mechanism. Thus, host and ether loads are also taken into account in our study. Our measurements highlight the current ultimate bounds on performance which may be achieved by user process applications communicating across machines, and serve as a guide in designing performance critical applications. For this study, hosts and ethers have been loaded with a user defined mix of tasks, i.e., an artificial workload. Moreover, we present a detailed timing analysis of the dynamic behavior of the TCP/IP and the UDP/IP network communication protocols' current implementation in Berkeley UNIX 4.2BSD. This study sheds light on the tradeoffs encountered when software and hardware perform the same actions on data, e.g., checksums, and when several buffering schemes coexist at different levels in the system.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**39**	

A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX † 4.2BSD

Luis Felipe Cabrera ‡
Edward Hunter
Mike Karels
David Mosher

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA

Abstract

Berkeley UNIX 4.2BSD is an operating system which provides alternative ways for processes to communicate with each other. User processes may choose, for example, intermachine communications media, protocols, addressing families, and styles of communication. In particular, user processes may use datagram or stream communication. In this paper we present a study of the impact that IPC mechanisms, as currently implemented in Berkeley UNIX 4.2BSD in Ethernet based environments, have, from the user process viewpoint, on the performance of distributed applications.

This study not only assesses the impact that different processors, network hardware interfaces, and Ethernets have on the communication across machines between user processes, but also the effect of the loading of the various components which participate in the interprocess communication mechanism. Thus, host and ether loads are also taken into account in our study. Our measurements highlight the current ultimate bounds on performance which may be achieved by user process applications communicating across machines, and serve as a guide in designing performance critical applications. For this study, hosts and ethers have been loaded with a user defined mix of tasks, i.e., an artificial workload.

Moreover, we present a detailed timing analysis of the dynamic behavior of the TCP/IP and the UDP/IP network communication protocols' current implementation in Berkeley UNIX 4.2BSD. This study sheds light on the tradeoffs encountered when software and hardware perform the same actions on data, e.g., checksums, and when several buffering schemes coexist at different levels in the system.

Index Terms: Berkeley UNIX, benchmarking, interprocess communication, datagram, virtual circuit, TCP protocol, UDP protocol, IP protocol, Ethernet, artificial workload, dynamic program profile.

1. Introduction

Users of distributed computing environments are faced with the issue of their optimal utilization. The availability of extensive pools of resources for their cooperative action presents new

† UNIX is a Trademark of AT&T Bell Laboratories

‡ On leave from the Departamento de Ciencia de la Computación of the Escuela de Ingeniería of the Pontificia Universidad Católica de Chile.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Goverment.

challenges for users of such facilities. Not only are users confronted with very powerful computing environments, but the possibility of parallelism may also lead them to revise their algorithms to determine whether this environment can help them achieve their computational needs faster, or more reliably.

Berkeley UNIX 4.2BSD is an operating system which provides alternative ways for user processes to communicate with each other [10-11, 19]. User processes may choose, for example, intermachine communications media, protocols [15-17], addressing families, and styles of communication. In particular, user processes may use datagram or stream communication. In Berkeley UNIX 4.2BSD two processes wishing to communicate need not have a common ancestor. In this paper we present a study of the impact that IPC mechanisms present in Berkeley UNIX 4.2BSD have, from the user process viewpoint, on distributed applications. We have studied the implementations which existed during the summer of 1984 in Ethernet based environments [13-14]. (The system we measured already differed in many aspects with the released 4.2BSD Berkeley UNIX.) Hereafter, we shall refer to this version of Berkeley UNIX 4.2BSD as the 'current' system. Results presented show the type of performance a user might expect from the communication mechanisms and medium when writing a distributed application.

In particular, one question investigated is the cost of sending data as datagrams on a local area network. Previous reports [20-21] indicate that Ethernets appear to be very reliable. These reports claim the actual loss of packets in the ether to be of one in two million, with current hardware interfaces. Thus, if reliable stream communication is too expensive, it may be desirable to base applications on datagrams, handling retransmissions of lost packets at the user level, to achieve better average performance.

Another point to be considered is the effect that the reliability built into the TCP protocol [17, 14] has on the user process perceived network latency and overall network performance. What is the penalty of using TCP compared to the cost of UDP if the underlying network is quite reliable? As some Ethernet interface hardware provide CRC checksums, should we use TCP in such an environment and have user applications penalized by this redundancy? If the reliability of the network is also high, then perhaps some other, lighter-weight protocol should be used; for instance, a sequenced packet protocol [24], or a protocol which does not require checksums when data is only being sent locally.

As there have been some questions about whether or not TCP/IP should be used in a local area network [14], our study will present, for Berkeley UNIX 4.2BSD, the actual costs incurred when doing so. The negative 'instinctive' reaction expressed by some, that TCP presents an unreasonable performance penalty for distributed computing in local area networks, may not be significant when one takes into account other issues involved in generating a TCP transmission from user space.

Lastly, for the range of network hardware available for this study we shall attempt to determine the effect the speed of the underlying network has on the user process' view of network performance. There are studies which suggest that certain ranges of communications media speed need not affect user applications, because of the limiting host processor speeds [9]. Thus, faster networks may increase the global volume of data transferred, by allowing more machines to coexist on the network before saturation is reached, but not affect the individual user process' perception of the network's performance.

The rest of this paper is subdivided as follows. In Section 2 we present the basic measurement assumptions. Section 3 has a complete discussion of the experimental environment used in our study. Section 4 presents basic network results when both hosts and ether are unloaded, while Section 5 reports on our measurements when there is load in the sending host and the receiving host. In Section 6 we study the effect of ether load. Section 7 contains a detailed timing analysis of the current TCP/IP and UDP/IP implementations. Finally, Section 8 consists of our conclusions.

2. The User Process Viewpoint

Local area network technology in the form of broadcast networks has long been with us. In particular the Ethernet [13], developed at Xerox PARC in the mid-seventies, has come into widespread use. Even though several studies have evaluated and modelled the performance of Ethernets under various conditions [1, 5-8, 18, 20-21], measurements performed in most of those studies have only determined the extreme limitations of this technology, and the degrees of performance degradation which can be expected at the lowest level, i.e., when analyzing communication performance from the hardware interfaces viewpoint. Indeed, only [7, 21, 23] have measurements of the performance a software system may expect. However, those measurements do not refer to communications between user processes but between operating system ones. In [9] we find an analysis which includes modelling the behavior of Ethernets under different parametric load conditions, in the context of file servers.

One aspect that has not been satisfactorily addressed in the literature is that of the performance the *user* of this type of local area network can expect. In this paper we focus our attention on the performance perceived by user processes communicating across machine boundaries. We call this the *user process viewpoint* of interprocess communication. It should be emphasized that the measurements being made from user space include all overhead caused by the protocol implementations and by the operating system.

When building a distributed application, a user can either use one of the system-supplied interfaces or implement a specialized set of network functions that support his application. The easier choice by far is using a system supplied interface. In Berkeley UNIX 4.2BSD, given the protocols supported and the styles of communication available (datagrams and stream communication), ad-hoc communications mechanisms seem to be necessary only for specialized tasks requiring very specific network services. We assert that these tasks do not normally fall in the user process category but really belong in one of the system processes. Thus, the evaluation of the system provided interfaces to the user processes is of interest. With this information, the user can decide how best to implement his application, and whether the performance requirements of the application are likely to be satisfied. Understanding how our local area network implementations perform makes it easier to design distributed applications. It also provides a better knowledge of what the current limitations of these applications might be, and where the improvements may come from.

3. The Experimental Environment

This paper describes a series of tests, performed at UC Berkeley, designed to determine the main performance properties of the existing Ethernet-based IPC mechanisms under Berkeley UNIX 4.2BSD. Our tests cover a wide range of packet sizes, host configurations, and network interfaces, within the context of the DARPA Internet protocols.

Berkeley UNIX 4.2BSD currently has only three protocols available for communication across processors. These are: TCP, UDP, and IP. Each of them is described in a separate document [15-17]. TCP/IP and UDP/IP provide different kinds of services to the user processes. Jointly, they provide a cross section of the minimum IPC services that should be expected to be available on any machine providing network access. However IP, as currently implemented, cannot be accessed directly by user processes (but can by super-user processes). Thus, we shall not deal explicitly with IP in this paper. It should be clear, however, that since UDP and TCP use IP, the performance of its implementation affects user process applications.

Of the several approaches which can be taken to assess intermachine IPC performance, we have chosen to instrument user code and execute specially written routines to stress different aspects of the network communication mechanisms. Therefore, *all* measurements were performed by user processes. Moreover, the routines written for this purpose were coded carefully so that we could minimize additional overhead. We have included samples of our test software in Appendix A.

3.1. Hardware Components Available for our Study

All our tests were performed using equipment available in the Computer Science Division at the University of California at Berkeley†. The Computer Science Division has a collection of DEC VAXes of various sizes and configurations, and a large number of SUN II workstations. Among the hosts used for this study, the VAX 11/780s were connected to a single 3 megabit/second Ethernet, the 11/750s to a 10 megabit/second Ethernet, and the SUN IIs also to a 10 megabit/second Ethernet. In addition, some of the VAXes were connected to a 10 megabit/second Ethernet using interface hardware from different manufacturers. Figure 1 illustrates that part of the network configuration that was available for our experiments. This collection of machines provides a partial cross-section of the hardware on which Berkeley UNIX 4.2BSD can run.

The machines selected for our tests are listed in Table 1. (Those hosts which had interfaces of different speeds appear twice.) The table shows both the physical memory size and network interface available on each machine. These machines were selected because they presented the widest variety of hardware configurations, and because we could control the system loads better. In other words, these machines could be used stand-alone. In Sections 3.2 and 4.1 we describe the software used for this study.

3.2. The Interprocessor Software

The software used for this study is divided into two groups: the one which measures the network throughput and latency, and the one which assesses the protocol implementations.

The network test software is based on two programs: a packet generation and a packet receiving program. The packet sending program transmits packets without expecting acknowledgement from the receiving program. It is used to determine the maximum speed at which the network hardware interfaces can be driven, and to generate traffic as fast as possible in the

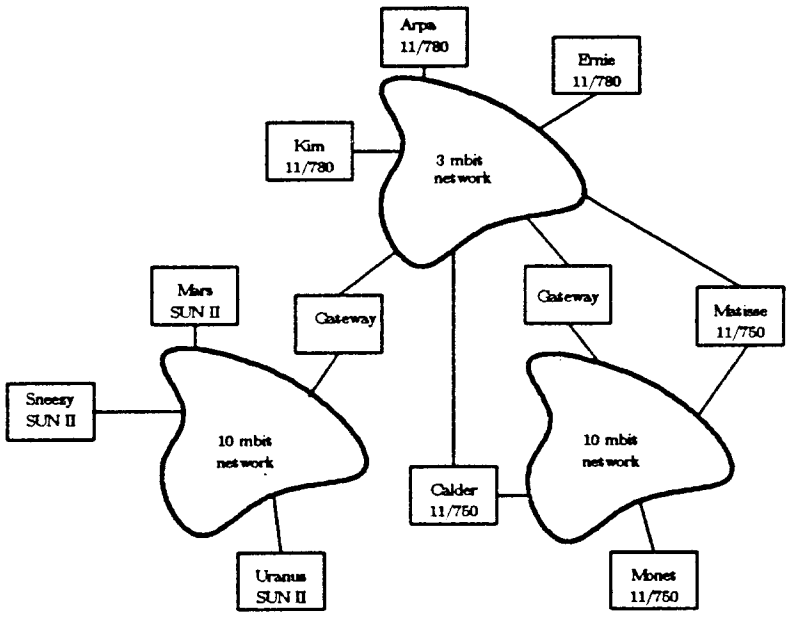

Figure 1: Partial Configuration of the Local Area Network

† We really thank the users of these facilities for their patience during our testing.

Host Name	Machine Type	Memory Size (megabytes)	Interface Speed (megabytes/sec.)	Interface Brand
ucbarpa	VAX-780	4	3	Xerox
ucbkim	VAX-780	4	3	Xerox
ucbcalder	VAX-750	2	3	Xerox
ucbcalder-10	VAX-750	2	10	3COM
ucbmatisse	VAX-750	1	3	Xerox
ucbmatisse-10	VAX-750	1	10	DEUNA
ucbmars	SUN II	2	10	3COM
ucburanus	SUN II	2	10	3COM

Table 1: Installation Configurations

hope of obtaining back-to-back packets at the receiving interface. An 'end of transmission' packet is sent at the end of a sending cycle. ‡

The packet receiving program acts as a sink server, i.e., it receives packets and drops them by going back to a 'read' state after each reception. This program keeps track of the number of packets received and of the time spent in doing so. It is also able to recognize an end of transmission packet. The sink server is particularly important for UDP/IP (and IP) assessment, since it provides a verification that the packets to be sent have actually left the sending host and have been received. Since we do not have a network hardware monitor, we need a software tool for this purpose.

Each of our testing programs was prepared in two versions, one for each of TCP/IP and UDP/IP. The time required to send one packet of data was determined by sending a large number of them, recording the hardware wall clock time for the total transmission, and obtaining the average time per packet. The user process timing was obtained using the *time* command.

The software used in Section 7 for assessing the implementation of TCP/IP and UDP/IP is based on profiling the kernel through the use of the commands *kgmon* and *gprof*, and on the use of an instrumented version of the kernel. A test program which sends a fixed amount of data a predetermined number of times was designed for each protocol. The kernel monitoring facility was enabled during the execution of the test program, which were run in single-user mode to avoid all possible interference. The profiles were determined later from the data collected by the monitor.

3.3. The Sizes of Messages

The seven user data packet sizes used for the network tests were: 1, 112, 224, 512, 1024, 1460, and 2032 bytes. This range of packet sizes was chosen to be representative of the type of traffic which might be present on a local area network [20-21], and also to exercise the buffer management schemes provided by the protocol implementations. The smallest size, 1 byte, was chosen to represent character-at-a-time user process transactions, and also to give an indication of the minimum user process message transmission delay, which we call user process network latency. The 112 byte size was chosen to represent individual lines of text, such as program listings or documentation, as might be output by some user program. It is also the case that 112 bytes is the maximum user data packet size that fits into one 128 byte 'mbuf' (in mbufs, 8 bytes are used for link pointers, 4 bytes for the data offset, 2 bytes for the size, and 2 bytes for the type). There are also 1 kilobyte mbufs. Mbufs are used inside the kernel for network software memory management. They are an integral part of the network software which manages its own

‡ It should be noted, though, that our 3 megabit/second Xerox network interfaces insert a delay, of the order of 1 millisecond, between the sending of consecutive packets to the same host. Thus, to stress a receiving interface, several hosts need to be concurrently sending packets.

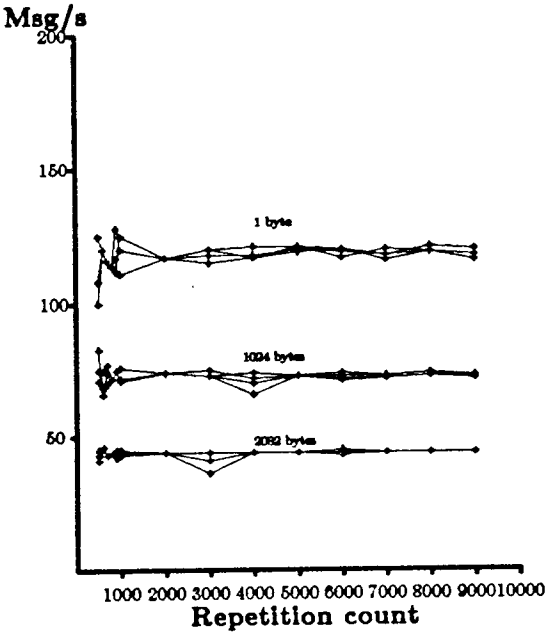

Figure 2: UDP/IP Datagrams/second versus Repetition Count

segment of the virtual memory. The 224 byte data size was chosen because it uses exactly two 128 bytes buffers. The 512 byte size corresponds to that of a common disk block which might be shipped around in user or file server applications (even though the current system tries to make this size invisible).

The 1024 byte size represents one logical page of data in Berkeley UNIX 4.2BSD; the networking software has been optimized for this size. Data which is not page sized is normally stored in chains of the smaller size buffers. Thus, a series of 'copy' instructions from user space into the system's small mbufs are required. The one kilobyte buffers are passed by the network protocol software to the network driver by simply augmenting a reference count, whereas the chains of the smaller buffers must be assembled into one contiguous buffer before being given to the network driver. Therefore, if the data is page sized, a copy operation can be avoided. In Section 7, the effect that saving one of these copy operations has on network performance will become apparent.

The 1460 byte size is the maximum packet size supported by the Ethernet interface hardware. (In fact, the maximum transmittable packet size is 1500 bytes, but 40 bytes are used by the ethernet protocol for header information.) Message fragmentation begins at this point. The 2032 byte size is the maximum buffer space allocated by the system to any one connection. It should be noted that, as the network software automatically fragments two kilobyte packets, the 2032 byte packet size also tests the effect of IP packet fragmentation on network performance.

It should be remarked that, for every message to be sent, the network software always assigns a 128 byte mbuf for the 40 byte header that is required by the TCP/IP and UDP/IP protocols. In the case of a 1024 byte user packet, all the additional bytes associated with it are appended to the 40 byte header information in that 'companion' 128 byte mbuf. To preserve data page alignment, the small mbuf is piggybacked onto the large one for transmission through the network, using trailer protocols [17] to avoid assembly copying.

3.4. Determining the Repetition Count

Each network test consisted of sending a fixed size packet a predetermined number of times. We call 'repetition count' the number of times each packet was sent. A repetition count of 4,000 was chosen for the networking tests. For the dynamic microanalysis of the protocol implementation, however, a repetition count of 10,000 was chosen.

A large number of preliminary UDP/IP 'transmit only' sessions were performed to determine the appropriate repetition count to be used. The objective was to find a repetition count which would be large enough so as to give us confidence in the timing results, yet small enough so that the tests could be performed in a reasonable amount of time. The preliminary tests were run using the 10 megabit/second hardware, in which we had absolute control over all other ether traffic. This eliminated the possibility of interference from other hosts while our tests were running. A series of repetition counts for packets of different sizes were tried. Each test was run five times. Thus, each point which appears in the figures of this section is the average of five sample points. Moreover, to eliminate the possibility of unknown system interference, the benchmark tests were also performed with the sending and receiving hosts interchanged. The results were found to be identical. In the figures of this and the following sections, all graphs represent plots of discrete points. The connecting lines interpolate linearly the results to highlight any trends which might exist. The packet sizes chosen for the preliminary exploration were 1 byte, 1024 bytes, and 2032 bytes. The repetition counts tried were 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, and 9000.

In [8], where a version of Section 4 of this paper was presented, it was observed that for UDP/IP, the total transmission time, TTT, as a function of the repetition count, RC, was characterized by

$$TTT = m*RC + n ,$$

where for 1 byte datagrams the slope 'm' was 33/6000, for 1024 byte datagrams it was 41/6000, and for 2032 byte datagrams it was 64/6000. The larger the message, the larger the slope. Having observed this transmission stability for large repetition counts, what we needed to determine

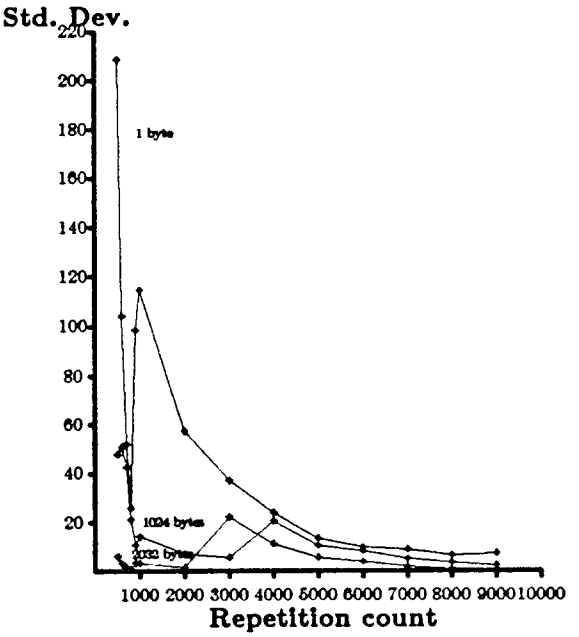

Figure 3: Standard Deviation of the Sample of UDP/IP Datagrams/second versus Repetition Count

was the minimum repetition count which would yield reliable measurements.

Figure 2 displays, for each of our three packet sizes, the number of UDP/IP datagrams per second sent as a function of the repetition count. We see that repetition counts of 4,000 and larger produce very stable results. Figure 3 has, for each of our three packet sizes, plots of the standard deviation of the samples used in Figure 2 as a function of packet size. It is clear from it that repetition counts below 2,000 behave in a fairly erratic way. Repetition counts above 4,000 produce very stable results. Our choice of 4,000 as the repetition count came as a compromise between length of the experimental runs and precision of the results.

For protocol assessment, the approach was to run our test software stand-alone. Here, the higher the repetition count, the larger the degree of accuracy we could obtain from our profiling tools. To obtain values accurate to one-tenth of one millisecond, a repetition count of 10,000 was necessary.

3.5. The Artificial Workload Used

Once we had selected a repetition count, host loading and network loading had to be specified and generated. Loading of the hosts was accomplished through the use of a script which has been utilized in several evaluation studies [2-4] by one of the authors. Experience has proven it to be a balanced set of tasks. The script consists of a sequential series of commands which exercise different aspects of the system. It includes a C compilation, the running of a cpu intensive job, a text formatting job, a short editor session, the copying to the same disk of a 60,000 byte file, and UNIX commands which request status information from the system, such as *ps -alx* and *who*. The original script [3] was modified so that it would continually cycle through its tasks without sleeping. In an otherwise idle host, we defined the unit of 'processor load' to be one copy of this script executing. Thus, a load of four units is obtained by having four copies of the script executing simultaneously (note that their start-up times were not staggered). To avoid file naming conflicts, each of the copies of the script was run from within a separate directory.

The unit of load on the ether was determined after an analysis of the case when both the hosts and the ether were idle. We used as unit of 'ether load' the traffic generated by the continuous sending into the ether of 1024 byte packets using TCP/IP (this unit of load is on the order of 750,000 bits/second). To carry out an experiment with an ether load of three, for example, we needed a total of eight machines. Six hosts pairwise transmitting data between themselves, while hosts seven and eight exchanged data between them as in the case of an unloaded ether.

4. Measurements with Unloaded Hosts and Ethers

In this section we present the results of our study with unloaded hosts and ether. They were performed using the computer systems described in Table 1, and they are summarized in Table 2. Some of them appear in [8].

4.1. The Family of Software Experiments

Two major groups of tests were performed corresponding to the two protocols: TCP/IP, and UDP/IP. Each test involved sending messages to another host using one of the protocols available. Each individual test was performed five times, and the results analyzed to insure that the underlying load conditions had not changed. The measurements were then averaged for display. Because of the unsocial hours at which the testing was performed, the test software was the only software running except for various system daemons. The system daemons woke up occasionally to perform system management functions, but this extra load was considered insignificant. In any case, it represents the expected load when a server is idle on a Berkeley UNIX 4.2BSD system running a single application, like a file server. The command *la* was executed between consecutive runs to verify that the system activity did not change drastically during any of the test runs.

In addition, the *netstat* UNIX command was used at the beginning and end of each run to determine whether collisions or network hardware interface errors were ever a distorting factor in

Test #	Test Type	Host -> Host	
1	Send	Calder-10	Matisse-10
2	Send	Matisse-10	Calder-10
3	Send	Calder	Matisse
4	Send	Matisse	Calder
5	Send	Arpa	Kim
6	Send	Kim	Arpa
7	Send	Mars	Uranus
8	Send	Uranus	Mars
9	Send	Arpa	Calder
10	Send	Calder	Arpa
11	Send	Calder-10	Calder-10
12	Send	Calder	Calder

Table 2: Test Configuration

any of the results. However, as other hosts were also connected to the ether being used, and because of the existence of gateways connected to these ethers, we had no effective control on the actual ether traffic. Lack of a dedicated network monitor prevented us from monitoring ether traffic during the runs. Because of the carrier sense nature of Ethernet networking, transmission delays due to the medium used clearly affect our measurements. These delays are not reported by *netstat*. Some of the variance observed in our measurements may be explained by the busy ether.

4.2. The 3 and 10 megabit/second Ethers

In Figures 4 through 7 we show our results for the case in which both the hosts and the ethers were not loaded. Those measurements obtained using a 10 megabit/second ether appear joined with a solid line, while those for a 3 megabit/second ether appear with a dashed line. The two labels on the curves represent the type of the sending and of the receiving host, respectively.

As seen from their specifications [15, 17], the UDP and TCP communication protocols differ essentially in that UDP does not guarantee reliable delivery, sequencing, duplication, or flow control. However, UDP does preserve datagram boundaries. TCP keeps a copy of each message in transit until a positive acknowledgement is received. TCP retransmits after a 'timeout' period if it has not received an acknowledgement. Only after a predetermined number of retransmissions have failed (10 in the Berkeley UNIX 4.2BSD implementation) does TCP return an error message to the user-level application which had requested the transmission.

The internal buffering strategies of these protocols differ. TCP tries to maximize packet size per send between hosts, whereas UDP cannot. Both UDP and TCP, however, do checksums at the sending end of the communication. Our implementation of UDP does not verify checksums at the receiving end of the communication, while TCP does so. Both of these protocols are considered 'heavy weight', because of the amount of robustness they possess, even though UDP is a lot less robust than TCP.

4.3. Analysis of These Results

This analysis is broken into two subsections. The first deals with network latency. The second, with the overall network performance achieved, in terms of throughput and elapsed transmission time per message size.

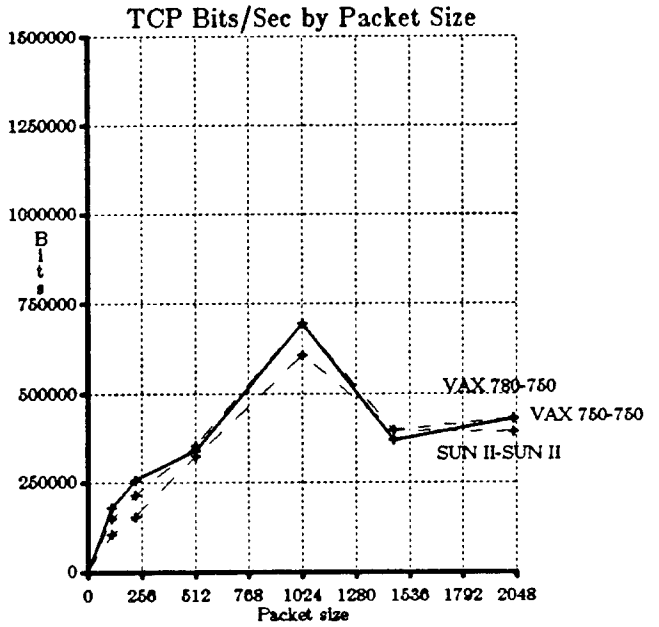

Figure 4: Bits/Second vs. Packet size. TCP protocol.
___ 10 megabit/second ether; ..... 3 megabit/second ether;

4.3.1. User Process Network Latency

The user process network latency is defined as the minimum cost to complete a 1 byte network transmission. Thus, latency is represented by the minimum time required to successfully

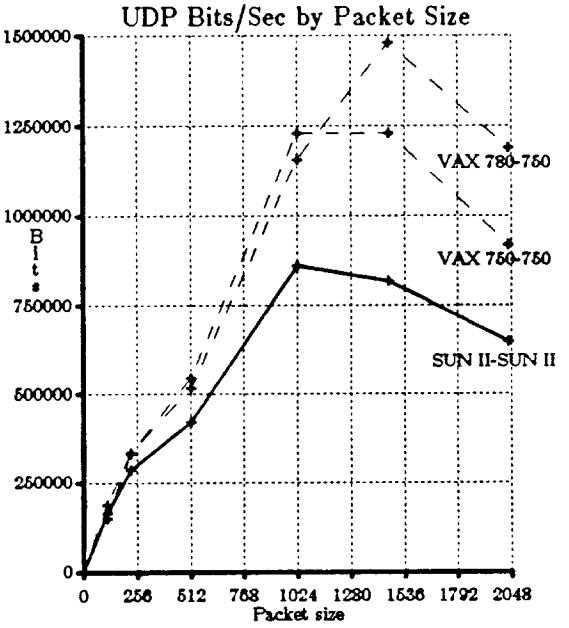

Figure 5: Bits/Second vs. Packet size. UDP/IP protocol.
___ 10 megabit/second ether; ..... 3 megabit/second ether;

send a single byte of data. In the case of our TCP runs, we include the cost of setting up only one connection per transmission session, as opposed to establishing an individual virtual circuit per message. The results show that, for the VAX 11/750, the latency for TCP/IP is approximately 5.5 milliseconds and for UDP/IP is 6.1 milliseconds (see the values for 'load 0' of Table 3 in Section 5.1, and Tables 1 and 2 of Appendix B). The results for the SUN II's indicate a latency of approximately 4.0 milliseconds for TCP/IP and 4.5 milliseconds for UDP/IP (see Tables 3 and 4 of Appendix B). A word of caution is in order here. We should keep in mind that, as TCP has flow control mechanisms, slow processing at the receiving end of a connection may cause buffering of transmission requests at the sending end. This is almost certainly the case with 1 byte user messages. Thus, the 'send only' experiments do not faithfully represent the minimum cost of a round trip of a few user bytes. The exact cost would be the one of establishing the connection, sending the bytes, receiving the acknowledgement, and closing the connection. Table 4 in Section 5.1 presents such measurements.

The results collected so far would seem to indicate that the network efficiency for our implementation of TCP/IP is greater than that for our implementation of UDP/IP. This is deceptive, however, due to the window-based flow control management which favors internal buffering of bytes by TCP when the receiving host restricts input flow. Because of this, for packet sizes less than 128 bytes, TCP/IP appears to be faster than UDP/IP. (This can be seen in Tables 1 and 2 of Appendix B by examining the seconds/transmission column. In each case TCP/IP gradually loses to UDP/IP, even though the first two kilobytes of data transmitted in each session probably take longer than the rest of the transmission due to buffering in the sending end of the connection. The crossover point appears to be around 128 bytes.)

On a VAX 11/750 this latency appears to fall between 5.5 and 6.5 milliseconds, depending on which hardware interface and which protocol are being used. For example, the 10 megabit/second interface on Matisse (see Tables 1 and 2 of Appendix B) appears to be about a millisecond slower than the one on Calder (see Tables 2 and 5 of Appendix B). Unfortunately, there is no way in our experiments to trace the difference in performance to either the hardware or the device driver.

Other studies, most notably [5, 25], have performed tests to assess the network's latency. In each case the values assigned to network latency have been smaller than the values we have measured. Even though those measurements were performed in system space, whereas ours were in user space, our network latencies still seem high. Unfortunately, our software measurement tools do not point to a specific protocol layer or hardware interface as a possible bottleneck. It is a problem area for future study determining what activities are contributing most to the latency. Section 7 does shed light, though, on the kernel costs which are incurred by the current protocol implementations.

4.3.2. Network Throughput

An unexpected observation was that, for UDP/IP, the 3 megabit/second hardware appears to be faster than the 10 megabit/second hardware (see Tables 2 and 6 of Appendix B). The measured times of user to users transmissions for the 3 megabit/second interfaces are consistently lower than those for any of the 10 megabit/second interfaces. In fact, the maximum throughput (slightly less than one and a half megabit/second) was obtained using the 3 megabit/second hardware (see Figure 5). The maximum throughput observed for the 10 megabit/second Ethernets was on the order of 800,000 bits/second (see Figure 8). There appears to be no good explanation for this but that of interface hardware design. However, for TCP/IP (see Tables 1 and 7 of Appendix B) both interfaces appear to have very similar speeds. This may be caused by the processors becoming the bottleneck when processing the TCP/IP packets. We have yet to verify this hypothesis, but Figure 8 is consistent with it.

The user-perceived throughput of the network does not increase linearly with packet size. This is obvious in the UDP/IP results, where throughput continues to climb until the largest

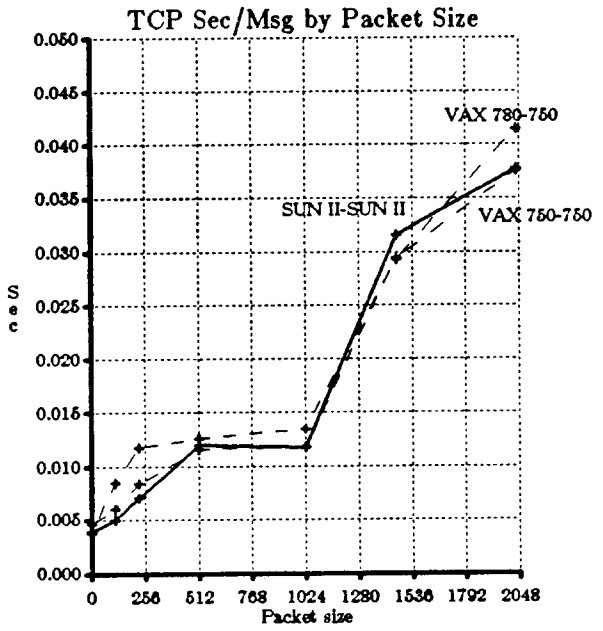

Figure 6: Seconds/Message vs. Packet size. TCP protocol.
___ 10 megabit/second ether; ..... 3 megabit/second ether;

possible packet is sent. At this point it falls off, as shown in Figures 5 and 8. One explanation of this phenomenon is the start of IP fragmentation beyond the 1460 byte packet size. The cost of fragmentation outweighs any performance gained by using larger buffers within the system. In fact, packet fragmentation has a very negative performance impact.

In comparing the results for TCP/IP and UDP/IP (see Figures 4 and 5), it is interesting to note that the difference in speed between the implementation of both protocols becomes more pronounced in the region around 1460 byte packets. There is a definite dip in the throughput for TCP/IP at this point, as can be seen in Figure 4, for both types (3 megabits/second and 10 megabits/second) of network interfaces. This divergence between them can be explained by the time the implementation of TCP requires to copy each data packet from an internal retransmission queue. The UDP/IP protocol does not have to pay this penalty, because it does not guarantee reliable delivery of packets. Its throughput continues to climb until internetwork fragmentation becomes a factor. Our implementation of TCP, on the other hand, must copy the same data three times: the first from user space to system space, the second from the transmission queue to the network interface, and the third in the network driver to make it contiguous. As more data per packet is transmitted, an increase in throughput is expected, as shown for the sizes below 1024 bytes. At the 1024 byte mark, two of the copy operations are eliminated by incrementing the reference counts on the page frames, causing a large positive jump in the throughput. Tables 5 and 6 of Section 7 show this in detail. Above 1024 bytes, all data is copied at the network driver level. This offsets the advantages gained by the virtual page swap done for 1024 bytes, and, again, as larger packets are sent, the throughput increases.

Figures 6 and 7 display the hardware wall clock time of transmitting messages of different sizes. This data is clearly related to the throughput data, given our fixed repetition count. We may see, for example, that it takes on the order of 0.012 seconds to transmit a 1 kilobyte message using TCP/IP, and between 0.007 and 0.009 seconds to do it through UDP/IP, depending on the sending and receiving hosts. As these measurements were obtained when there was no load on the ether or on the systems, these timings need to be considered lower bounds for the wall clock time. One should note that, for both TCP/IP and UDP/IP, the wall clock time for transmitting a 1

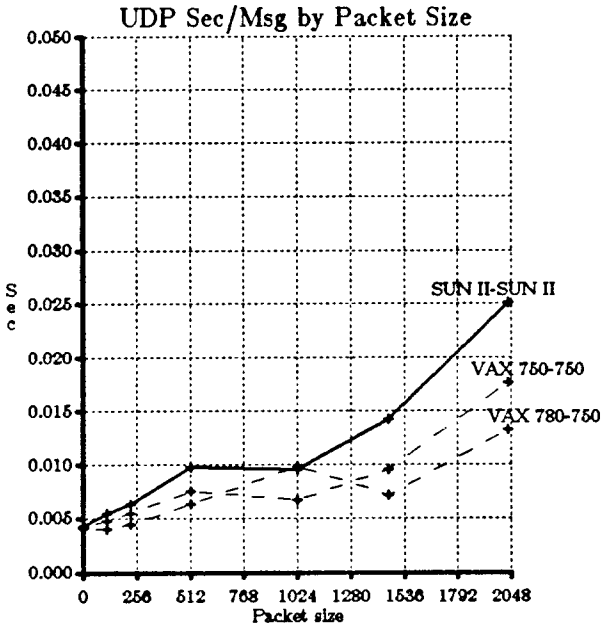

Figure 7: Seconds/Message vs. Packet size. UDP/IP protocol.
___ 10 megabit/second ether; ..... 3 megabit/second ether;

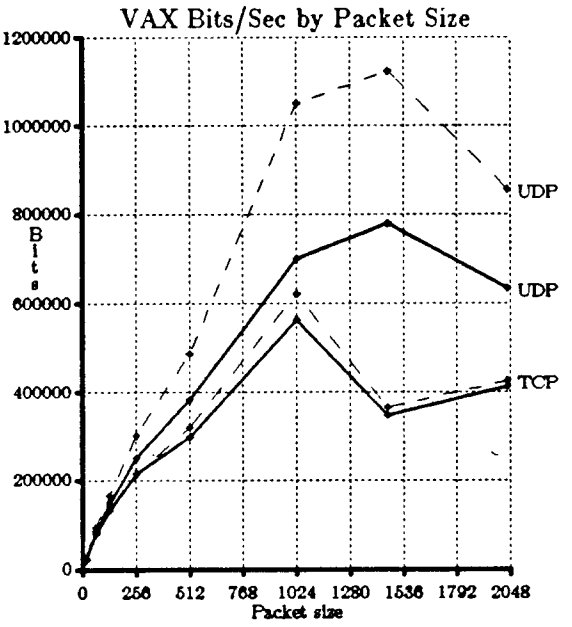

Figure 8: Bits/Second vs. Packet size.
___ 10 megabit/second ether; ..... 3 megabit/second ether;

kilobyte message is only twice as much as that of transmitting a 128 byte message. Transmitting a 512 byte message is as costly as transmitting a 1 kilobyte message. Tables 5, 6, 7, and 8 of Section 7 explain this phenomenon in terms of the internal copying that the implementations of the protocols do in their buffer management. Messages which are smaller than 1 kilobyte get copied into multiple 128 byte mbufs. Indeed, given this buffer management scheme, transmitting a 1023

message takes about 60% more time than transmitting a 1024 byte message.

Figure 8 presents the result of performing tests 1 and 3 of Table 2 for TCP/IP and UDP/IP. We may appreciate a substantially different behavior by the two network protocols. While TCP/IP's throughput shows rather small changes, we see that UDP/IP's throughput does vary substantially. Most remarkable is the fact that the higher throughput, for both protocols, is achieved through the **slower** ether. This means that the 10 megabit/second network hardware interfaces we have are slower than the 3 megabit/second ones. The minimal throughput variation for TCP/IP could be explained in terms of processor saturation. It is clear that, because of the retransmission processing, TCP/IP is more cpu demanding than UDP/IP. Figure 8 proves that distributed applications designers need, unfortunately, to be aware of the specific hardware characteristics of the networking subsystem of the installations where the applications will run, especially if their applications are time-critical ones.

5. Measurements with Loaded Hosts and Unloaded Ether

The results of this section have been obtained from measurements done on only two configurations of sending and receiving hosts: SUN II to SUN II, and VAX 11/750 to VAX 11/750. As described in Section 3.5, the artificial workload used consisted of running multiple copies of a script in an otherwise idle machine, while generating and receiving network traffic.

Sections 5.1 and 5.2 show that the impact the cpu load has on network throughput is of such a magnitude, that the amount of information for the user process level that may be obtained from those studies where only the unloaded case is presented is very small.

5.1. Loaded Sender

Figures 9 through 12 present our results for the case where the 'sender' host was loaded, and where both the receiver host and the ether were unloaded. SUN results are presented with dashed lines while VAX results are presented with a solid line. Those measurements corresponding to processor load 0 were already displayed in Section 4.2.

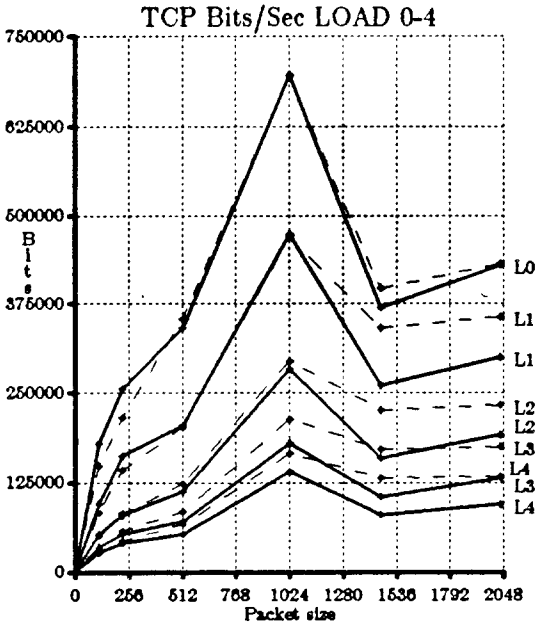

Figure 9: Bits/Second vs. Packet size. TCP/IP.
..... VAX 11/750; ___ SUN II;

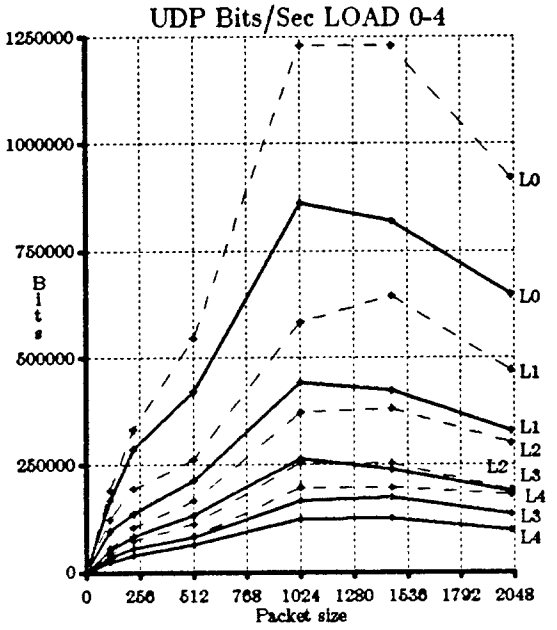

Figure 10: Bits/Second vs. Packet size. UDP/IP.
..... VAX 11/750; ___ SUN II;

Figures 9 and 10 show the substantial effect that load has on throughput for both kinds of hardware configurations, and for both protocols. (Notice that, for clarity, the vertical scales do not coincide in Figures 9 and 10.) Network throughput for both protocols, as perceived by a user level application, is almost halved for each successive level of our artificial load in the range 0-2. This ratio is observed for both architectures: SUNs and VAXes. Given the multiple-user orientation of VAXes, it seems clear that user applications running on non-dedicated machines will perceive fairly low levels of network throughput. It is somewhat reassuring to see, though, that the VAX architecture appears to handle slightly better than the SUN architecture the degradation at higher levels of load. For packet sizes above 1024 bytes, TCP on the VAX handles itself somewhat better than in the SUN (see Figure 9).

Processor	SUN II		VAX 11/750	
Load	TCP/IP	UDP/IP	TCP/IP	UDP/IP
load 0	4.0	4.5	5.5	6.1
load 1	7.9	9.2	8.5	8.0
load 2	13.9	18.0	14.6	14.0
load 3	20.1	21.5	22.8	21.0
load 4	25.0	30.5	29.3	27.6

Table 3: Network Latencies in Milliseconds for Loaded Sender and Undloaded Receiver.

It is also interesting to see, for TCP/IP and UDP/IP, the 'flattening' of the throughput curves for higher values of load. While one can see seven-fold increases in throughput for different packet sizes at low levels of load, for loads 3 and 4 these ratios are much lower. In fact, at high loads, user processes will perceive, for TCP/IP and UDP/IP, an almost constant throughput rate across message sizes: the user process cost for sending 112 bytes or 2048 bytes will be almost

Size	VAX 11/750 TCP/IP				
	load 0	load 1	load 2	load 3	load 4
1 byte	18.5	21.5	27.0	28.5	50.2

Table 4: Round Trip Time in Milliseconds Between two VAX 11/750 for 1 byte under TCP/IP. Loaded Sender and Unloaded Receiver.

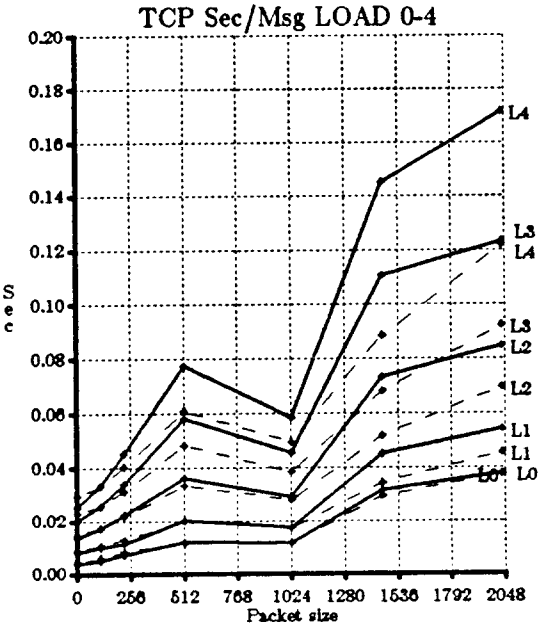

Figure 11: Seconds/Message vs. Packet size. TCP/IP.
..... VAX 11/750; ___ SUN II;

equal, and the advantages of sending 1024 byte messages will not be much.

Figures 11 and 12 present the time, in seconds per message, that it takes to transmit a packet of the specified size in each of the protocols. In obvious accordance with throughput decrease, packet transmission times also increase substantially with the load. From Figure 11 we see that, for TCP/IP, a rather 'linear' behavior of the transmission times can be observed throughout all values of load. Of course there is the 'dip' at 1024 bytes, which, as was explained in Section 4.3.2, corresponds to a savings in copying between the protocol and the network interfaces. For UDP/IP, on the other hand, it is seen how, from a rather 'linear' behavior of the transmission time for load 0, we go towards a more 'exponential' behavior for higher loads. The SUN measurements indicate that for UDP/IP SUNs degrade more substantially than VAXes.

No protocol seems to be substantially better off than the other one in terms of reduced throughput degradation with load. However, Figures 9 and 10 show that, in the SUN's case, the throughput advantage that UDP/IP had over TCP/IP at load 0 practically vanishes for all other levels of load. On SUN's not processing in dedicated mode, user space applications would see no difference between protocols.

Table 3 displays the data for user process network latency under loaded sender. Just as the throughput was affected, latency also increased substantially. In spite of TCP/IP internal buffering, we see more than a six-fold increase in the SUNs and a five-fold increase in the VAX.

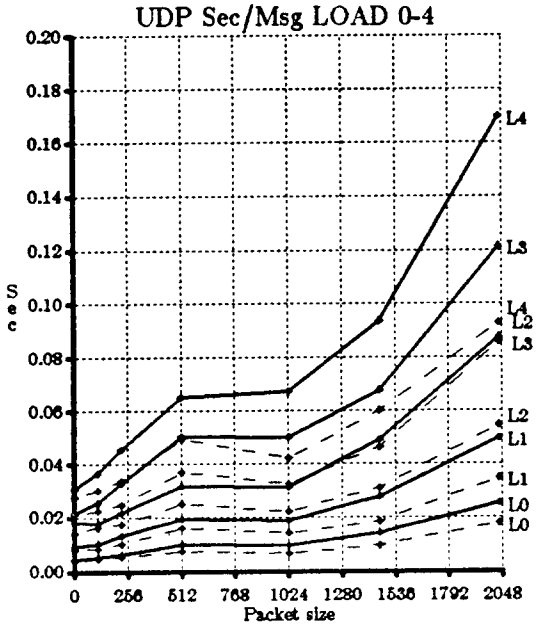

Figure 12: Seconds/Message vs. Packet size. UDP/IP.
..... VAX 11/750; ___ SUN II;

The implementation of UDP/IP had a very similar degradation behavior. Table 4 displays stand-alone measurements over a 3 megabit/second Ethernet of the round trip of 1 byte using TCP/IP between two VAX 11/750 (each measurement is the average of three sample points). The sending host was loaded as before, while the receiver (and loop-back) host was idle. The ether was otherwise idle. These measurements are indicative of the time a user process would require to receive a short reply to a short request. They complement the results of Table 3.

5.2. Loaded Receiver

For this subsection we only have VAX 11/750 based results. In Figures 13 through 16 we present them for the case where the 'receiver' host was loaded, and both the sender and the (3 megabit/second) ether were unloaded. Those measurements corresponding to processor load 0 were already displayed in Section 4.2.

For TCP/IP, as was the case before, we see the severe impact of the load on throughput and message transmission times. If the protocol interfaces need to drop a packet at the receiving end, because the host had no time to service it before new data arrived, the sending host will retransmit the message. Thus, all messages sent are received. By comparing Figures 13 and 9, one can appreciate that the effect of the load on TCP/IP's throughput is slightly less in the case of a loaded receiving host, than it is on the sending host. This can be explained by pointing out that sending and receiving are not symmetrical TCP operations. TCP does fewer operations with the data at the receiving end, e.g., it does not have to queue messages for retransmissions. Comparing Figures 15 and 11 we may appreciate the differences in transmission times.

As UDP/IP does not guarantee reliable delivery of messages, one does not expect load on the receiving host to affect the user process perceived sending throughput. This is absolutely confirmed by Figure 14. However, the amount of packet loss is nonnegligible. User level applications using UDP/IP should plan to keep some accounting of the messages sent, if they care, to offset the effect that a loaded receiver has on the network traffic. Figure 15 displays the seconds/message in this case.

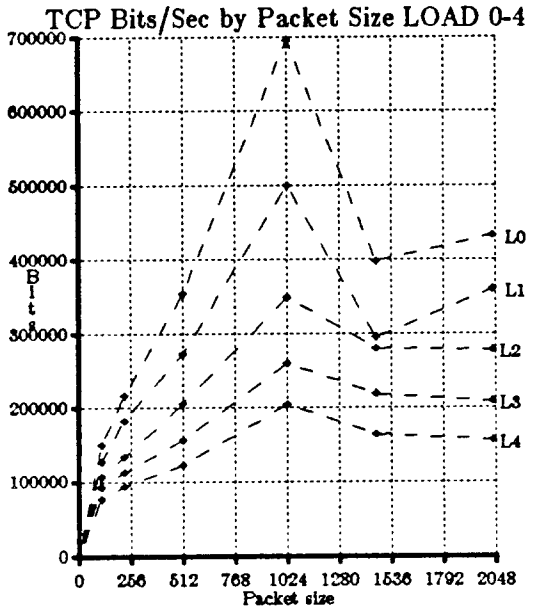

Figure 13: Bits/Second vs. Packet size. TCP/IP. VAX 11/750.
Unloaded Sender and Loaded Receiver.

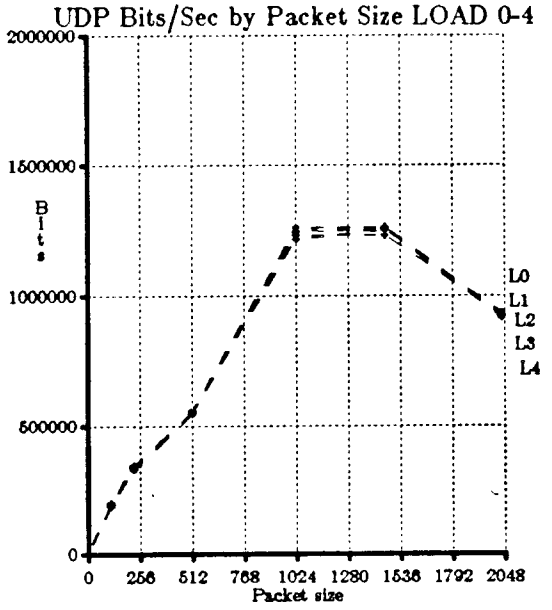

Figure 14: Bits/Second vs. Packet size. UDP/IP. VAX 11/750.
Unloaded Sender and Loaded Receiver.

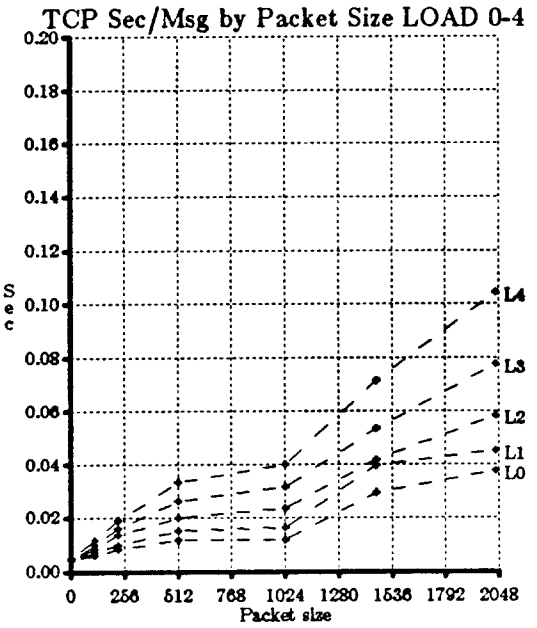

Figure 15: Seconds/Message vs. Packet size. TCP/IP. VAX 11/750.
Unloaded Sender and Loaded Receiver.

5.3. Loaded Sender and Loaded Receiver

Measurements with both sender and receiver hosts loaded were not very different from those with only one of the hosts loaded. Figure 16 displays measurements obtained in an otherwise empty ether with the sending and receiving hosts' loads represented in the labels by L0 or L1. A pair Lx-Ly means that the sending host had load x while the receiving host had load y. We may appreciate that the presence of a load on both hosts did not produce differences of more than 20% with respect to the cases where only one of the hosts was loaded. With the sole exception of the 1460 byte size, the L1-L1 case had the lowest throughput.

6. Measurements with Loaded Ether

To generate the desired ether load, we used TCP/IP transmissions between pairs of otherwise idle SUNs. Load three was achieved by having three pairs of SUNs communicating, while two other SUNs established the communication to be measured. Figure 16 displays our results for the cases with ether load 0 and ether load 3. The traffic generated by each unit of ether load was on the order of 0.75 megabit/second; thus an ether load of three guarantees a 2.2 megabit/second traffic. This is 22% of the medium's capacity. The Ethernet used for these measurements has more than 30 SUNs on it, but, because of the late hours when our runs were made, the rest of the traffic was very light.

For this level of ether load, the throughput degradation was on the order of 40%. Lack of a hardware monitor makes us speculate that not only the carrier sense mechanism prevented transmissions (at least 22% of them in the average), but also there were collisions which further degraded throughput. It is interesting to note that at this level of ether load we can observe almost no network discrimination for different packet sizes. The only possible exception could be the 512 byte size.

7. Assessment of Protocol Implementation

In this section we study the implementations of TCP/IP and UDP/IP in detail. The primary goal of this study is to understand the specific costs of using TCP/IP and UDP/IP as

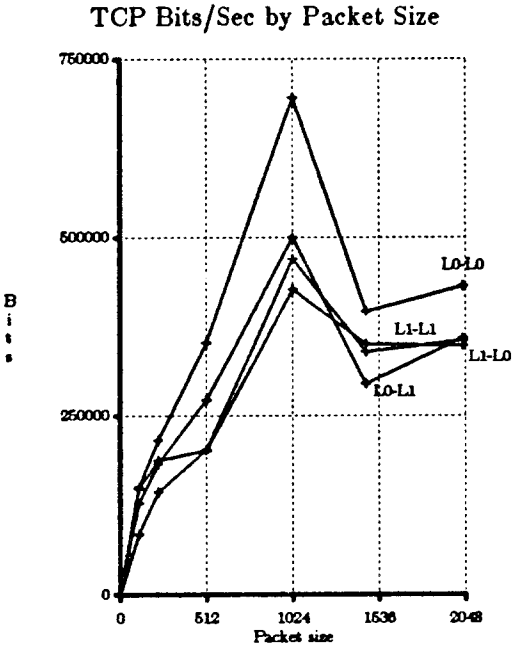

Figure 16: TCP/IP Transmissions Between two VAX 11/750.
3 megabit/second Ethernet.

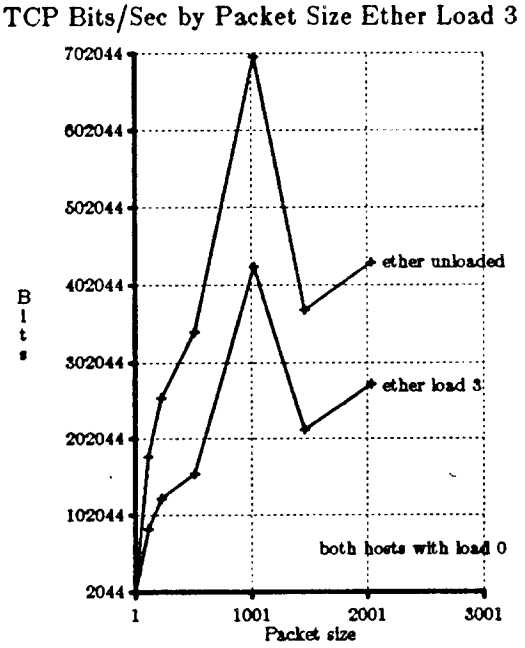

Figure 17: TCP/IP Transmissions Between SUN II.
10 megabit/second Ethernet Unloaded and with Ether Load 3.

currently implemented. Secondarily, this study may point out unexpected performance penalties, or inappropriate software 'optimizations'.

To study the implementations of TCP/IP and UDP/IP, we have employed user level processes to exercise the kernel, and the UNIX commands *kgmon* and *gprof* to provide an execution profile for the kernel. Two similar test programs have been designed, one for each protocol,

which send a specified number of messages of a specified length to a predetermined host exactly 10,000 times. In each run, from the profiled kernel we obtained the execution history of sending a message with the specified amount of data. The kernel profiling facilities were enabled only during the actual running of each test program. The test programs were run in single user mode to avoid interferences as much as possible. The test programs have been included in Appendix A.

These tests were run between two VAX 11/750's over a 3 megabit/second ether (see Table 2). One processor was used to profile the kernel while the test program ran; the other processor just sinked the data from the test programs. The test programs sent a set of six different amounts of data per message: 1, 112, 113, 1023, 1024, and 1025 bytes. These amounts were chosen to illustrate boundary conditions of the network data buffer management system.

It must be remarked, however, that lacking a hardware monitor to measure message size distribution, and not having instrumented the kernel so as to have an appropriate distribution of packet sizes sent per host, we do not have, at this point, an absolute way of judging the effectiveness, for the user process applications, of the current protocol implementations. What we do have, however, is an excellent breakdown of kernel time spent while sending messages of the chosen sizes, and a detailed knowledge of what would happen if, say, a user space application would sent messages of any given size.

The repetition count of 10,000 was chosen to achieve measurement accuracy of one tenth of one millisecond per message. A greater level of accuracy could be achieved with a larger repetition count, but the testing time for performing such accurate experiments has not been available.

The raw data from these profiles appears in the following subsections. The values represent the number of seconds spent in each routine during the 10,000 transmissions. Because these values were obtained from a single run for each message size, they are mostly intended to show the relative ordering of the routines with respect to time utilization, and to show gross changes in the magnitude of time utilization of each routine. Obtaining absolute timings would require further data stability analysis. In Tables 5 and 6 we have highlighted in boldface those routines which exhibit larger timing variations as a function of the amount of data to be processed.

For both protocols, the buffer scheme used in the implementation appears to have an overshadowing effect on performance. Since UDP/IP sends data atomically, and only limits a packet's maximum size, this protocol is not sensitive to varying data sizes. The drastic increases in overhead in the routines shown in bold appear to be due to the data buffer management scheme chosen. On the other hand, TCP/IP, with its windows and data streaming, is sensitive to the amount of data presented. Thus, in addition to the increased overhead seen in the routines which deal with data within the buffer management system, the actual protocol overhead appears to increase noticeably because of the varying amounts of data presented.

7.1. TCP/IP

Table 5 presents the time spent in a selected group of routines that are called to process a message with a specific amount of data. These values were obtained directly from the *gprof* output.

The calling hierarchy for sending data via TCP/IP starts with a system call, *syscall*, to a generic *write* operation. *write* calls the generic data transfer routine *rwuio*, which transfers no data. In turn, *rwuio* calls the specific routine which can perform the necessary operation for the type of object, in this case *soo_rw*. *soo_rw* calls the appropriate internal routine that implements the original request to 'send data', *sosend*. *sosend* is responsible for copying the data from the user space via *uiomove*, which calls *Copyin* to do the actual copying. *sosend* first determines the amount of buffer space available for this specific socket, and then copies the minimum of the buffer space available or the amount of data to be sent, whichever is smaller, into internal buffers. These buffers are then passed to the appropriate protocol, in this case *tcp_usrreq*, which switches immediately to *tcp_output*, the output sequencer for the TCP protocol. At this level (*tcp_output*), the specific amount of data is copied from the TCP output queue by *m_copy*. These data buffers

TCP/IP Routines and System Calls	Packet Size					
	1	112	113	1023	1024	1025
syscall	2.91	2.79	2.76	3.02	3.01	3.12
write	0.72	0.81	0.81	0.78	0.84	0.90
rwuio	1.57	1.67	1.82	1.77	1.84	2.23
soo_rw	0.92	0.83	0.85	0.82	0.77	0.84
sosend	**3.89**	**4.76**	**6.24**	**15.53**	**5.53**	**16.52**
uiomove	**0.99**	**1.02**	**1.77**	**9.23**	**1.38**	**8.85**
Copyin	**0.45**	**1.17**	**1.68**	**10.96**	**6.86**	**11.19**
ipintr	**0.19**	**0.93**	**0.94**	**6.31**	**4.88**	**5.11**
tcp_usrreq	2.20	2.20	1.93	2.00	2.03	2.24
tcp_input	**0.06**	**1.32**	**1.67**	**11.14**	**10.68**	**10.97**
tcp_output	**6.26**	**6.16**	**6.34**	**8.50**	**7.14**	**11.23**
sbappend	**1.65**	**1.63**	**2.11**	**8.33**	**1.11**	**8.02**
ip_output	2.78	3.07	3.14	3.36	3.35	5.63
tcp_cksum	**2.23**	**3.13**	**3.65**	**16.16**	**10.52**	**16.72**
in_cksum	**1.89**	**1.72**	**1.57**	**3.69**	**2.66**	**4.41**
m_copy	**2.77**	**3.86**	**5.22**	**18.24**	**2.38**	**29.73**
enoutput	2.74	3.38	3.45	3.09	4.15	5.08
enstart	2.87	2.53	2.81	2.17	2.78	4.53
if_wubaput	**3.83**	**4.00**	**5.30**	**14.23**	**4.70**	**16.58**
in_lnaof	2.44	2.21	2.23	2.76	2.72	3.43

Table 5: Partial Decomposition of TCP/IP Time in Seconds
for 10,000 Transmissions Between to Dedicated VAX 11/750.
[*Highlighted in boldface are those calls with larger timing changes.*]

UDP/IP Routines and System Calls	Packet Size					
	1	112	113	1023	1024	1025
syscall	3.48	3.02	3.40	3.29	2.76	3.67
sendto	0.84	0.94	0.91	0.83	0.99	0.85
sockargs	0.80	0.93	0.87	0.81	0.77	0.84
getsock	0.79	0.62	0.46	0.63	0.64	0.71
sendit	2.92	2.84	2.94	2.85	2.69	2.46
m_freem	**2.59**	**2.35**	**2.98**	**3.20**	**4.43**	**2.28**
useracc	0.56	0.55	0.55	0.60	0.63	1.03
sosend	**5.07**	**5.75**	**6.68**	**19.25**	**5.58**	**7.38**
uiomove	**1.20**	**1.48**	**2.05**	**10.66**	**1.34**	**2.52**
Copyin	**1.14**	**2.02**	**2.37**	**14.58**	**8.45**	**8.47**
udp_usrreq	1.83	1.90	1.52	2.00	1.56	1.91
in_pcbconnect	2.24	2.45	2.13	2.56	2.65	2.17
in_pcblookup	0.95	1.13	0.81	1.28	0.96	1.03
in_netof	2.67	2.42	3.16	3.27	3.13	3.33
if_ifonnetof	0.46	0.46	0.53	0.64	0.73	0.62
in_pcbdisconnect	0.52	0.64	0.53	0.55	0.55	0.46
udp_output	2.61	2.45	2.28	3.66	3.07	2.78
m_get	**1.96**	**1.46**	**1.86**	**1.93**	**3.54**	**2.06**
ip_output	4.12	3.89	3.53	4.26	4.01	4.14
udp_cksum	**2.23**	**3.37**	**2.40**	**12.82**	**8.79**	**8.28**
in_cksum	4.19	4.54	4.43	3.44	3.56	4.01
in_lnaof	2.25	2.41	2.43	2.44	2.40	2.27
ipintr	4.42	3.71	3.96	2.69	4.34	4.36
enrint	3.07	3.51	4.44	3.37	3.90	4.14
enoutput	3.36	3.12	3.04	3.46	3.87	2.80
enstart	3.16	2.88	2.50	2.66	2.02	2.67
if_wubaput	**4.02**	**4.31**	**5.08**	**14.96**	**4.01**	**10.54**
getf	0.39	0.36	0.41	0.27	0.48	0.44

Table 6: Partial Decomposition of UDP/IP Time in Seconds
for Sending 10,000 Datagrams Between to Dedicated VAX 11/750.
[*Highlighted in boldface are those calls with larger timing changes.*]

are placed on the data queue for the TCP connection. Based on the protocol and its windowing techniques, an amount of data to be sent is selected and then passed to the IP level, *ip_output*. Data and header are checksummed in *tcp_cksum*, and later the additional IP header information is checksummed in *in_cksum*. Finally, the message is queued and possibly sent to the specific network interface for transmission. In this study, the network interface is represented by the two functions *en_output* and *en_start*. Before such a transmission can happen, the buffered data must be copied into a single contiguous memory space; this is done in *if_wubaput*.

From the row entries in Table 5, we can see that, of the 20 different routines listed for TCP/IP, 11 present processing costs which vary significantly with the amount of data sent. The processing time of the other 9 routines remains practically constant. The four calls which show a larger variation in the vicinity of the 1024 bytes region are *sosend*, *uiomove*, *m_copy*, and *if_wubaput*. All are associated with the buffer management strategy. The five calls which have a larger impact in the processing of messages are *tcp_output*, *sosend*, *if_wubaput*, *tcp_cksum*, and *m_copy*. Clearly, the greatest impact comes from those routines which do copying of data within the interfaces. In the 1024 case, checksumming and servicing acknowledgements and window updates through *tcp_input* are the two most expensive tasks.

As mentioned in Section 1, there are network hardware interfaces which provide checksumming facilities. As can be observed from the entries for *in_cksum* and *tcp_cksum*, the time spent in it is substantial. This, in fact, is true for both protocols (see Tables 5 and 6). It is clear, then, that redundant checksumming in a system has definite performance penalties.

7.2. UDP/IP

From the user process viewpoint, sending data through UDP/IP results from system calls equivalent to *sendto*. *sendto* requires the destination address on each call; *sockargs* and *getsock* produce the socket control block associated with this operation. With this information, *sendit* is called, which in turn calls *sosend* as TCP/IP does. Again *sosend* calls *uiomove*, which calls *Copyin* to actually copy the user data into kernel buffers. These buffers are given to *udp_usrreq* which calls *udp_output* after a pseudo connection is established via *in_pcbconnect* with its associated routines, *in_pcblookup, in_netof, and if_ifonnetof*. As with TCP, *udp_output* represents the output processing of the UDP protocol. At this level the header is created. The header and the data are then passed to *ip_output* as in TCP/IP. *ip_output* checksums the header in *in_cksum* and passes the message to the appropriate network interface, in this case *en_output*. Again, the mbufs must be copied into a single contiguous memory space before transmission; this is done in *if_wubaput*.

From the row entries in Table 6, we can see that of the 28 different routines listed for UDP/IP, only 7 present processing costs which vary significantly with the amount of data sent. The processing time of the other 21 routines remains practically constant. One exception to this is *m_freem*, which exhibits its largest processing time for datagrams of size 1024 bytes. The two calls which show a larger variation in the vicinity of the 1024 bytes region are *uiomove*, as before

TCP/IP Routines and System Calls	Packet Size 1	112	113	1023	1024	1025
syscall	10,882	10,884	10,882	10,884	10,882	10,882
write	*10,002*	*10,002*	*10,002*	*10,002*	*10,002*	*10,002*
rwuio	10,009	10,010	10,009	10,010	10,009	10,009
soo_rw	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
sosend	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
uiomove	**10,014**	**10,015**	**20,014**	**100,015**	**10,014**	**97,158**
ipintr	**356**	**1,532**	**1,686**	**10,035**	**10,238**	**10,012**
tcp_usrreq	10,011	10,020	10,021	10,119	10,022	10,046
tcp_input	**323**	**1,472**	**1,645**	**10,095**	**10,243**	**10,012**
tcp_output	10,010	10,017	10,015	10,104	10,008	10,015
sbappend	10,000	10,000	10,000	10,088	10,000	10,000
ip_output	**10,016**	**10,049**	**10,032**	**10,182**	**10,024**	**20,081**
tcp_cksum	**10,333**	**11,497**	**11,671**	**20,234**	**20,256**	**30,037**
in_cksum	**10,384**	**11,628**	**11,730**	**20,494**	**20,356**	**30,369**
m_copy	**10,007**	**10,022**	**10,024**	**10,136**	**10,010**	**20,022**
enoutput	**10,016**	**10,049**	**10,032**	**10,182**	**10,024**	**20,081**
enstart	**10,024**	**10,059**	**10,037**	**10,188**	**10,057**	**20,141**
if_wubaput	**10,016**	**10,049**	**10,032**	**10,182**	**10,024**	**20,056**
in_lnaof	**30,075**	**30,190**	**30,131**	**30,644**	**30,136**	**60,361**

Table 7: Number of Calls per Function for 10,000 TCP/IP Transmissions.
[*Highlighted in boldface are those calls with large variations.*]
[*Highlighted in italics are those calls with identical counts.*]

UDP/IP Routines and System Calls	Packet Size					
	1	112	113	1023	1024	1025
syscall	10,886	10,888	10,886	10,888	10,886	10,886
sendto	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
sockargs	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
getsock	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
sendit	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
m_freem	**19,003**	**19,406**	**19,204**	**19,331**	**29,194**	**19,848**
sosend	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
uiomove	**10,015**	**10,016**	**20,015**	**100,016**	**10,015**	**20,015**
udp_usrreq	*10,002*	*10,002*	*10,002*	*10,002*	*10,002*	*10,002*
in_pcbconnect	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
in_pcblookup	10,052	10,055	10,049	10,089	10,069	10,073
in_netof	40,057	40,051	40,075	40,108	40,075	40,111
if_ifonnetof	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
in_pcbdisconnect	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
udp_output	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
m_get	**20,023**	**20,021**	**20,030**	**20,040**	**30,029**	**20,041**
ip_output	10,019	10,017	10,025	10,036	10,025	10,057
udp_cksum	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
in_cksum	29,021	28,820	28,458	28,710	29,818	29,770
in_lnaof	30,108	30,105	30,122	30,196	30,143	30,183
ipintr	8,996	9,403	9,211	9,347	9,888	9,850
enrint	8,255	8,503	8,298	8,369	8,117	8,136
enoutput	10,019	10,017	10,025	10,036	10,025	10,037
enstart	10,037	10,039	10,037	10,056	10,028	10,044
if_wubaput	10,019	10,017	10,025	10,036	10,025	10,037
getf	*10,840*	*10,840*	*10,840*	*10,840*	*10,840*	*10,840*

Table 8: Number of Calls per Function for 10,000 UDP/IP Datagrams.
[*Highlighted in boldface are those calls with large variations.*]
[*Highlighted in italics are those calls with identical counts.*]

with TCP/IP, and *if_wubaput*. The first is associated with the buffer management strategy, and the latter with passing the data to be transmitted to the hardware interface in one contiguous piece. The five calls which have a larger impact in the processing of datagrams in UDP/IP are *udp_cksum*, *sosend*, *if_wubaput*, *uiomove*, and *Copyin*. For UDP/IP, checksumming is, across most datagrams sizes, the single most expensive operation performed. As mentioned in Section 7.1, redundancy of this operation should be avoided. (TCP cannot avoid it.) For larger datagrams sizes, the greatest impact comes from those routines which do copying of data across the interfaces. *sosend*, as was also the case in TCP/IP, is an important factor in the time spent processing messages.

Table 6 also shows that for UDP/IP the processing costs are somewhat more evenly distributed across many routines than for TCP/IP. This suggests that speeding up UDP/IP will require streamlining many routines. In UDP/IP there appear not to be many significant gains to be obtained from any one optimization.

Another way to look at these profiling results is to consider the number of times each routine was called in the processing of the 10,000 transmissions. Table 7 presents, for TCP/IP, such a decomposition, while Table 8 has it for UDP/IP. The most surprising behavior observed was for

uiomove in TCP/IP, where for the 1025 packet size, the count, instead of dropping to something in the order of 20,000, remains very high. This is so because of the 'stream' communication nature of TCP. As there are no record boundaries, the use of 'odd' send sizes causes both fragmentation and joining of segments. If there was more than 1 kilobyte of data to be transmitted but less than 1 kilobyte of buffer remaining for the socket, *sosend* would send the remaining amount using small mbufs. This is exactly what happened in the 1025 byte case as witnessed by *uiomove* which is called once for each mbuf used. It is interesting to note that the UDP/IP implementation is more immune than the TCP/IP implementation to packet size changes, with respect to the number of times individual routines are called. This is mostly due to the fact that UDP preserves record boundaries and thus may allocate mbufs with better knowledge. In the UDP/IP implementation, however, *m_freem* and *m_get* exhibit a peak of activity for the 1024 byte case which contrasts with all other sizes. This is due to the handling of trailer protocol packets.

8. Conclusions

For users who want to implement distributed applications based on Berkeley UNIX 4.2BSD computing environments interconnected through Ethernets, the system currently provides two basic transmission protocols for interprocess communication: TCP and UDP. Both, in turn, are based on IP for actual data transmissions. They are part of the DARPA Internet family of protocols. This paper has presented results which show the network performance a user process may expect from the network when implementing distributed applications. Even though there are currently other protocol families at different stages of implementation which will coexist with the above protocols in the kernel of Berkeley UNIX, this paper has only addressed performance issues relating to TCP/IP and UDP/IP.

We have first determined that, for achieving an adequate degree of confidence in our networking measurements, a repetition count of 4,000 was necessary. When assessing the protocol implementations, however, we needed a repetition count of 10,000 to achieve an accuracy to the one tenth of one millisecond, even though these measurements were obtained executing in standalone mode. Network throughput and latency were obtained by sending messages between hosts by user space processes. We also added artificial loads to the sending and receiving hosts, and to the ether, respectively, to assess the effect of a load on throughput and packet transmission times. For protocol implementation evaluation, we had to use a version of the kernel compiled for profiling.

Given the variety of hardware at our disposal, we have been able to exercise many alternative communication paths, in different tests, throughout the months in which this study was performed. Two unexpected results related to hardware differences were that the 10 megabit/second interface on Matisse appears to be about a millisecond slower than the one on Calder (see Tables 1 and 2, and Tables 2 and 5 of Appendix B), and that the 3 megabit/second hardware appears to be faster than the 10 megabit/second hardware (see Tables 2 and 6 of Appendix B). Software overhead may also play a role here, in that the device on Calder is more complicated than the one on Matisse. However, for TCP/IP (see Tables 1 and 7 of Appendix B), both interfaces appear to have the same speed. This may be caused by the processors becoming saturated processing the TCP/IP packets.

We have defined user process network latency to be the minimum time required to send a single byte of data. When the ether, and both the sending and receiving hosts had no other user activities in them but our tests (what we call host load zero), our results show that, for the VAX 11/750, the latency for TCP/IP is approximately 5.5 milliseconds, and for UDP/IP is 6.1 milliseconds (see Table 3, and Tables 1 and 2 of Appendix B). For the SUN II's, we observed a latency of approximately 4.0 milliseconds for TCP/IP, and 4.5 milliseconds for UDP/IP (see Tables 3 and 4 of Appendix B). However, the round trip of a 1 byte TCP/IP transmission took 18.5 milliseconds between two unloaded VAX 11/750 (see Table 4).

Through the use of an artificial workload, we have seen the severe effect which the load has on the user processes' perception of network throughput. In the case of a loaded sender, not only network latency increases between five to six-fold (see Table 3), from a processor load of zero to a maximum processor load of four, but also network throughput is reduced, at load four, to less than 20% of what it is at load zero (see Figure 9). At high levels of load, our measurements indicate that, across all message sizes, user process applications will approximately see a constant network throughput, which will be on the order of 250,000 bits per second. This is so because throughput curves seem to 'flatten' at high levels of load (see Figures 9 and 10). Analogous results were observed for TCP/IP in the case where the receiver host was loaded and both the sender host and the ether were unloaded (see Figure 13). On the other hand UDP, from the sending user process level, was immune to the presence of a load at the receiving end, see Figure 14. We estimate that packet losses, in this case, are non-negligible.

The maximum network throughput observed corresponded to the case where an otherwise idle VAX 11/780 sent packets to an otherwise idle VAX 11/750. For UDP/IP, we saw rates which, for 1460 byte packets, approximated 1,500,000 bits per second. TCP/IP and UDP/IP exhibited different throughput behavior as a function of packet size. While TCP/IP showed a 'dip' immediately above packet sizes of 1024 bytes (see Figures 4 and 5), UDP/IP did so only above 1460 byte packets, where internetwork fragmentation begins. The TCP/IP cost of copying messages from the transmission queue to the retransmission queue appears as a main cause of the TCP dip mentioned above.

When transmissions were made with an ether load of three, and both sending and receiving hosts were kept with host load zero, we could observe a degradation of 40% in throughput. This was uniform across packet sizes, so, at least at this level of ether load where at least 22% of the transport capacity was used, there was no medium discrimination against any packet size.

A detailed protocol implementation analysis has been presented for TCP/IP and UDP/IP. For TCP/IP, those routines which do the copying of data appear to make preponderant contributions to the total elapsed time (see Table 5). For UDP/IP, the single most expensive operation is the computation of the checksums (see Table 6). Network buffer management is also a factor, but lack of facilities to determine the size distributions of packets sent per host does not allow us to fully assess the quality of the buffer management policies and mechanisms at this time.

9. Epilogue

Since this study was conducted several changes have been made to the implementations of TCP/IP and UDP/IP, as well as to the buffer management policies and default buffer sizes. These changes will be present in future BSD releases. We highlight some.

The buffer size at the socket level is now a settable parameter. When increased to 8 kilobytes we observed an improved throughput for UDP/IP in the order of 20%. In TCP, a facility for buffering outstanding small packets to be sent has been added. This minimizes, in the case of receiver busy, the number of transmissions between it and any sender. *sosend* has been changed so as to align 1 kilobyte packets whenever possible. From the receiver end of transmissions, and having in mind that the processing of acknowledgements consumes a substantial amount of processor resources, a scheme for delayed acknowledgements has been incorporated. This scheme works best with larger socket buffer size. Routing has been enhanced to cache the last computed route; if two consecutive packets go to the same destination the route for the second need not be computed. This should improve throughput for large data transfers. However, a complete assessment of these changes has yet to be made.

10. Bibliography

[1] Almes, G. T., and Lazowska, E. D., "The Behavior of Ethernet-like Computer Communications Networks", Proceedings of the 7th Symposium on Operating System Principles, 1979, pp. 66-81.

[2] Cabrera, L. F., "A Performance Analysis Study of UNIX," Proceedings of the Computer Performance Evaluation Users Group 16th Meeting, CPEUG 80, NBS Special Publication 500-65, Orlando, Florida, October 1980, pp. 233-243.

[3] Cabrera, L. F., "Benchmarking UNIX: A Comparative Study," in Experimental Computer Performance Evaluation (D. Ferrari and M. Spadoni eds.) North-Holland, Amsterdam, Netherlands, 1981, pp 205-215.

[4] Cabrera, L. F., and Rodriguez-Galant, G., Predicting Performance in UNIX Systems From Portable Workload Estimators Based on the Terminal Probe Method, Report No. UCB/CSD 84/194, University of California, Berkeley, August 1984.

[5] Cheriton, D., and Zwaenepoel, W., "The Distributed V Kernel and its Performance for Diskless Workstations", Proceedings of the 9th SOSP, November 1983.

[6] Gonsalves, T. M., "Packet-Voice Communication on an Ethernet Local Computer Network: an Experimental Study", Proceedings of the SIGCOMM 1983 Symposium on Communication Architectures and Protocols, Austin, Texas, March 1983, pp. 178-185.

[7] Hagmann, R. B., "Performance Analysis of Several Backend Database Architectures", Ph.D. Thesis, Report No. UCB/CSD 83/124, University of California, Berkeley, August 1983.

[8] Hunter, E., "A Performance Study of the Ethernet Under Berkeley UNIX 4.2BSD", Proceedings of CMG XV, San Francisco, California, December 1984, pp. 373-382.

[9] Lazowska, E. D., et. al., "File Access Performance of Diskless Workstations", Technical Report 84-06-01, June 1984, Department of Computer Science, University of Washington.

[10] Leffler, S. J., and Fabry, R., "A 4.2BSD Interprocess Communication Primer", Report No. UCB/CSD 83/145, University of California, Berkeley, July 1983.

[11] Leffler, S. J., et. al., "4.2BSD Networking Implementation Notes," Report No. UCB/CSD 83/146, University of California, Berkeley, July 1983.

[12] Luderer, G. W. R., et. al., "A Distributed UNIX System Based on a Virtual Circuit Theory", Proceedings of the Eighth Symposium on Operating Systems Principles, Asilomar Conference Grounds, Pacific Grove, California, December 1981, pp. 160-168.

[13] Metcalfe, R. M., and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks", CACM, Volume 19, Number 7, July 1976, pp. 395-404.

[14] Padlipsky, M., "TCP-ON-A-LAN", RFC 872, USC Information Sciences Institute, September 1982.

[15] Postel, J., "User Datagram Protocol", RFC 768, USC Information Sciences Institute, August 1980.

[16] Postel, J., "Internet Protocol - DARPA Internet Program Protocol Specification", RFC 791, USC Information Sciences Institute, September 1981.

[17] Postel, J., "Transmission Control Protocol", RFC 793, USC Information Sciences Institute, September 1981.

[18] Rajaraman, M. K., "Performance Measures for a Local Network", ACM Sigmetrics Performance Evaluation Review, Volume 12, Number 2, Spring-Summer 1984, pp. 34-37.

[19] Sechrest, S., "Tutorial Examples of Interprocess Communication in Berkeley UNIX 4.2BSD", Report No. UCB/CSD 84/191, University of California, Berkeley, June 1984.

[20] Shoch, J. F., and Hupp, J. A., "Performance of an Ethernet Local Network -- A Preliminary Report", Proceedings of Spring COMPCON 80, San Francisco, February 1980.

[21] Shoch, J. F., and Hupp, J. A., "Measured Performance of an Ethernet Local Network", CACM, Volume 23, Number 12, December 1980, pp. 711-721.

[22] Swinehart, D., McDaniel, G., and Boggs, D., "WFS: A Simple Shared File System for a Distributed Environment." Proceedings of the 7th SOSP, *Operating Systems Review*, Vol. 13, No. 5, pp. 9-17.

[23] Terry, D., and Andler, S., "Experience With Measuring Performance of Local Network Communications", IBM San Jose Research Laboratory Research Report RJ 3743 (43119), December 1983, pp. 1-6.

[24] Xerox Corporation, "Level Two: Sequenced Packet Protocol", Xerox XNS Protocol Handbook, May 1981.

[25] Walker, B., "The LOCUS Distributed Operating System", Proceedings of the 9th SOSP, November 1983.

11. Appendix A: Software Used in The Study

Test Software Used in This Study

11.1. Software for Network Performance Assessment

```
/* (@(#)Create.c 1.11            /vd/osmosis/systems/ker/lib/sccs/s.Create.c) */

#include <sys/param.h>
#include <sys/dir.h>
#include <stdio.h>
#include <sys/socket.h>
#include "in.h"
#include <sys/time.h>
#include <netdb.h>
#include <errno.h>

/* some useful definitions */
#define TRUE     1
#define FALSE    0

#define X_OK     1

/*  Parameters for test */
#define DOMAIN   AF_INET
#define DOMAINNAME        "AF_INET"
#define MAXMESSAGESIZE    4096
long    time ();
int     errno;

main (argc, argv)
char   *argv[];
int     argc;
{
    struct hostent *hp,
                   *gethostbyname ();
    struct sockaddr_in  name;
    struct sockaddr_in  cname;
    struct sockaddr_in  cname1;
    int     parentsock;
    int     ctlsock;
    char    run_name[20];
    char    donemsg;
    char    msgbuff[MAXMESSAGESIZE];
    int     repcount,
            messagesize,
            srepcount;
    int     namelen,
            cnamelen;
    int     readmask,
            writemask,
            excepmask;
    int     lost,
            got,
            sent;
    int     found;
    int     rv;
    struct timeval  timeout;
    long    starttime,
            endtime,
            ttime;
    int     i;
    long    bytecnt,
            bitcnt;
```

```
    long    bysec,
            bisec;
    float   delay;
    int     msec;
struct{
    int repcount;
    long ttime;
    long bysec;
    long bisec;
    int msec;
    float delay;
}childata;
    FILE * logfile;

    bysec = 0;
    bisec = 0;
    delay = 0.0;
    msec = 0;
    lost = 0;
    donemsg = 5;
    cnamelen = sizeof (cname);
    if (argc != 5) {
        fprintf (stderr, "usage: %s from-host to-host repcount msgsize0,
                argv[0]);
        exit (-1);
    }
    repcount = atoi (argv[3]);
    messagesize = atoi (argv[4]);
    sprintf (run_name, "ts%-d", messagesize);
    bytecnt = messagesize * repcount;
    bitcnt = bytecnt * 8;
    srepcount = repcount;
    fprintf (stderr,
            "from HOST: %s to HOST: %s REPCOUNT: %d, MESSAGESIZE: %d0,
            argv[1], argv[2], repcount, messagesize);
    if ((repcount <= 0) ||
            (repcount > 1000000) ||
            (messagesize <= 0) ||
            (messagesize > 4096)) {
        fprintf (stderr, "bad parm, rep count must be > 0, < 10000000);
        fprintf (stderr, "     and  msgsize must be > 1, <= 40960);
        exit (-1);
    }

    if (NULL == (logfile = fopen ("tsend.log", "a"))) {
        perror ("fopen");
        exit (-1);
    }

    if ((ctlsock = socket (DOMAIN, SOCK_STREAM, 0)) < 0) {
        fprintf (stderr, "error (%d) making kernel socketpair.0, errno);
        exit (-1);
    }
 /* bind it so we can get messages */
    hp = gethostbyname (argv[1]);
    bcopy (hp -> h_addr, &(name.sin_addr.s_addr), hp -> h_length);
    name.sin_family = AF_INET;
    name.sin_port = 0;
    if (bind (ctlsock, &name, sizeof (name))) {
        perror ("binding");
        close (ctlsock);
        exit (-1);
    }
 /* bind it so we can get messages */
    hp = gethostbyname (argv[2]);
    bcopy (hp -> h_addr, &(cname.sin_addr.s_addr), hp -> h_length);
```

```
    cname.sin_family = AF_INET;
    cname.sin_port = 2066;
    if (0 != connect (ctlsock, &cname, sizeof (cname))) {
        perror ("connect");
        close (ctlsock);
        exit (-1);
    }

    if ((parentsock = socket (DOMAIN, SOCK_STREAM, 0)) < 0) {
        fprintf (stderr, "error (%d) making kernel socketpair.0, errno);
        exit (-1);
    }
/* bind it so we can get messages */
    hp = gethostbyname (argv[1]);
    bcopy (hp -> h_addr, &(name.sin_addr.s_addr), hp -> h_length);
    name.sin_family = AF_INET;
    name.sin_port = 0;
    if (bind (parentsock, &name, sizeof (name))) {
        perror ("binding");
        close (parentsock);
        exit (-1);
    }
    hp = gethostbyname (argv[2]);
    bcopy (hp -> h_addr, &(cname.sin_addr.s_addr), hp -> h_length);
    cname.sin_family = AF_INET;
    cname.sin_port = 2076;
    if (0 != connect (parentsock, &cname, sizeof (cname))) {
        perror ("connect");
        close (parentsock);
        exit (-1);
    }

/* clear out buffer */
    for (i = 0; i < messagesize; i++) {
        msgbuff[i] = 1;
    }
    starttime = time (0);
    fprintf (stderr, "parent starting at %s", ctime (&starttime));
    while (1) {
        rv = send (parentsock, msgbuff, messagesize, 0);
        if (rv < 0) {
            fprintf (stderr, "parent exiting.  send code %d, errno %d lost %d sent %d got %d", rv, errno, lost, sent
            shutdown (parentsock, 2);
            exit (rv);
        }
        repcount--;
        sent++;
        if (repcount <= 0) {
            endtime = time (0);
            msgbuff[0] = 10;
            rv = send (parentsock, msgbuff, 1, 0);
            ttime = endtime - starttime;
            rv = send(ctlsock, &donemsg, 1,0);
            sleep(1);
            rv = recv(ctlsock, &childdata, (sizeof childdata), 0);
            shutdown (parentsock, 2);
            shutdown (ctlsock, 2);
            if (ttime != 0) {
                bysec = bytecnt / ttime;
                bisec = bitcnt / ttime;
                delay = (float) ttime / (float) srepcount;
                msec = srepcount / ttime;
            }
            fprintf (stderr, "parent exiting.  time = %d0hroughput:%ld bytes/sec %ld bits/sec %f sec/msg %d msg/sec0
                    bisec, delay, msec);
            fprintf (logfile, "%10s %10s %10s %d %d %d %ld %ld %d %f ",
```

```
                run_name, argv[1], argv[2], ttime, repcount, messagesize,
                bysec, bisec, msec, delay);
            fprintf (logfile, "%ld %ld %ld %d %f %d0, childdata.ttime,
              childdata.bysec, childdata.bisec, childdata.msec, childdata.delay,
              childdata.repcount);
            fclose (logfile);
            exit (0);
        }
    }
}
```

11.2. Software for TCP/IP Assessment

```
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in address;
char sendbuf[1025];;

main(argc,argv)
char *argv[];
{
        int i;
        struct hostent *phostent;
        int des;
        int size;

        des = socket(AF_INET,SOCK_STREAM,0);
        if (des < 0 )
                perror("socket"),exit(-1);
        phostent = gethostbyname(argv[1]);
        address.sin_family = AF_INET;
        address.sin_addr.s_addr  = *(int *)phostent->h_addr;
        address.sin_port = 4321;
        if (connect(des,&address,sizeof(address)))
                perror("connect"), exit(-1);
        size = atoi(argv[2]);
        for (i=0; i<10000; i++)
                write(des,sendbuf,size);
}
```

11.3. Software for UDP/IP Assessment

```
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in address;

main(argc,argv)
char *argv[];
{
        int s, i;
        char buf[1025];
        struct hostent *phostent;
        int size;
        extern errno;

        if((s = socket(AF_INET,SOCK_DGRAM,0))  == -1) perror("socket") ;
        phostent = gethostbyname(argv[1]);
        address.sin_family = AF_INET;
        address.sin_addr.s_addr  = *(int *)phostent->h_addr;
        address.sin_port = ntohs(1234);
        size = atoi(argv[2]);
        printf("Testing byte size of %d0,size);
        for(i = 0; i < 10000; i++)
                sendto(s,buf,size,0,&address,sizeof(address));
        if (errno) perror("udp: ");
}
```

12. Appendix B: Selected Raw Data

Tables With Raw Data Used For The Figures in Section 4

calder-10 -> matisse-10				
Bytes/ Message	Total Time	Transmissions/ Second	Bits/ Second	Seconds/ Transmission
1	11	181	1454	0.0055
16	11	172	22108	0.0058
64	13	153	78769	0.0065
128	15	128	131413	0.0078
256	19	104	213422	0.0096
512	27	72	299072	0.0137
1024	29	68	565234	0.0145
1460	67	29	347722	0.0336
2032	78	24	413917	0.0393

Table 1: TCP/IP sending VAX 11/750 10 megabit/second Ethernet

calder-10 -> matisse-10				
Bytes/ Message	Total Time	Datagrams/ Second	Bits/ Second	Seconds/ Datagram
1	12	163	1312	0.0061
16	12	158	20348	0.0063
64	13	153	78956	0.0065
128	14	140	144334	0.0071
256	16	121	249976	0.0082
512	21	93	383002	0.0107
1024	23	84	700949	0.0117
1460	30	66	779012	0.0150
2032	51	38	635038	0.0256

Table 2: UDP/IP sending VAX 11/750 10 megabit/second Ethernet

uranus -> mars				
Bytes/ Message	Total Time	Transmissions/ Second	Bits/ Second	Seconds/ Transmission
1	8	250	2000	0.0040
16	8	250	32000	0.0040
64	9	222	113777	0.0045
128	11	178	183078	0.0056
256	14	136	280868	0.0073
512	21	94	387566	0.0106
1024	23	83	689077	0.0119
1460	60	32	390033	0.0300
2032	70	27	463331	0.0351

Table 3: TCP/IP sending SUN II 10 megabit/second Ethernet

uranus -> mars				
Bytes/ Message	Total Time	Datagrams/ Second	Bits/ Second	Seconds/ Datagram
1	9	222	1777	0.0045
16	9	217	27875	0.0046
64	9	217	111501	0.0046
128	10	188	193628	0.0053
256	12	158	325578	0.0063
512	16	125	512000	0.0080
1024	17	115	953055	0.0086
1460	24	83	973333	0.0120
2032	42	46	770494	0.0211

Table 4: UDP/IP sending SUN II 10 megabit/second Ethernet

matisse-10 -> calder-10				
Bytes/ Message	Total Time	Datagrams/ Second	Bits/ Second	Seconds Datagram
1	13	146	1177	0.0068
16	14	142	18285	0.0070
64	14	140	72166	0.0071
128	15	126	129907	0.0079
256	17	117	240941	0.0085
512	20	97	397897	0.0103
1024	19	104	853692	0.0096
1460	24	82	965546	0.0121
2032	41	47	781647	0.0208

Table 5: UDP/IP sending VAX 11/750 10 megabit/second Ethernet

calder -> matisse				
Bytes/ Message	Total Time	Datagrams/ Second	Bits/ Second	Seconds/ Datagram
1	11	181	1459	0.0055
16	11	183	23487	0.0055
64	11	181	93090	0.0055
128	12	160	165414	0.0062
256	13	146	301573	0.0068
512	16	118	487905	0.0084
1024	15	128	1051306	0.0078
1460	20	96	1123504	0.0104
2032	38	52	855578	0.0190

Table 6: UDP/IP sending VAX 11/750 3 megabit/second Ethernet

calder -> matisse				
Bytes/ Message	Total Time	Transmissions/ Second	Bits/ Second	Seconds/ Transmission
1	10	192	1541	0.0052
16	11	183	23487	0.0055
64	12	163	84020	0.0061
128	15	129	133119	0.0077
256	19	105	216502	0.0095
512	25	77	320514	0.0128
1024	26	75	621984	0.0132
1460	64	30	365510	0.0320
2032	76	25	426765	0.0381

Table 7: TCP/IP sending VAX 11/750 3 megabit/second Ethernet